Lidando com Entidades

Site: Moodle institucional da UTFPR
Curso: CETEJ34 - Sistemas de Banco de Dados - JAVA_XXX (2024_01)
Livro: Lidando com Entidades

Impresso por: PAULO ROBERTO DOS SANTOS
Data: segunda-feira, 24 jun. 2024, 16:23

Índice

1. Lidando com Entidades

O conceito de entidades é bem simples de ser compreendido e nasceu nos bancos de dados relacionais. Todas as bases de dados são formadas por tabelas e estas, por linhas e colunas, onde cada linha é considerada uma entidade.

Em uma aplicação Java, preparada para trabalhar com um framework ORM, é preciso realizar o mapeamento objeto relacional. Este mapeamento é a forma como o framework vai referenciar uma tabela no banco de dados com a classe que a representa na aplicação.

E cada coluna da tabela vai ter uma referência na classe que será um atributo (variável de instância). Assim, cada classe da aplicação que referencia uma tabela no banco de dados é chamada de classe de entidade. Isto porque, uma classe vai ser a instância de um objeto que será persistido na tabela como uma linha, ou seja, uma entidade.

Os mapeamentos podem ser feitos de duas formas, uma delas, por meio de arquivo XML. Com o lançamento da versão 5 do Java anos atrás, a linguagem passou a oferecer o uso de anotações e elas foram adicionadas como opção de mapeamento pelos frameworks ORM e também pela especificação JPA.

Por exemplo, o Hibernate tem suas próprias anotações de mapeamento, se elas forem usadas, apenas aplicações que trabalham com o Hibernate farão uso deste mapeamento.

Já, se o mapeamento for realizado com as anotações oferecidas pela especificação JPA, qualquer framework ORM poderá fazer uso deste mapeamento. Sendo assim, é sempre mais apropriado usar o sistema de mapeamento da especificação.

1.1. Classe AbstractPersistable

Neste tópico serão abordados dois exemplos de classes de entidades mapeadas via anotações da especificação JPA.

Há dois motivos pelos quais preciso abordar as classes de entidades, o primeiro é que precisamos delas para trabalhar com os repositórios do Spring Data.

O segundo é referente a uma classe abstrata, fornecida pelo SpringData JPA, que minimiza a quantidade de código necessária para a implementação de cada classe de entidade.

Normalmente, este recurso é desenvolvido à parte, em projetos profissionais, o que o Spring fez foi oferecer uma implementação já pronta com o objetivo de facilitar as coisas para o programador.

A classe em questão é a **AbstractPersistable** e, seu uso é por meio de herança. Trabalhando com **AbstractPersistable,** uma classe de entidade vai herdar alguns métodos como **getId() e setId(),** os quais têm o atributo **id** já mapeado:

```
@Id
@GeneratedValue
private PK id;

public PK getId() {
   return id;

protected void setid(final PK id} {
   this.id = id;
}
```

Outro método interessante é o **isNew() ,** o qual verifica se o valor do atributo id é nulo ou não, caso seja, o retorno será verdadeiro, caso contrário será falso.

Este método é útil para saber, se uma entidade já foi persistida ou não. Lembre-se que toda entidade persistida já contém um identificador.

```
@Transient
public boolean isNew() {
   return null == getId();
}
```

O **toString()** também está presente em **AbstractPersistable,** assim como, os métodos **equals()** e **hashCode()**. Estes três métodos levam em consideração apenas o atributo id da entidade.

Se em algum momento for necessário ter uma implementação diferente em algum desses métodos citados, basta sobrescrevê-los na classe de entidade que herda estas características.

O pacote onde se encontra a classe **AbstractPersistable** é o **org.springframework.data.jpa.domain**.

Sobre a anotação @Transient:
Essa anotação é usada para indicar que um campo ou propriedade de uma entidade não deve ser persistido no banco de dados. Ou seja, quando anotamos um atributo de nossa classe entidade com **@Transient**, estamos dizendo ao provedor JPA que ele deve ignorar esse atributo durante o processo de persistência. Isso significa que o conteúdo desse atributo não será salvo no banco de dados ao persistir a entidade, e também não será recuperado do banco de dados ao carregar a entidade.

Essa anotação é útil em situações em que temos dados temporários ou cálculos que não precisam ser armazenados permanentemente no banco de dados. Pode ser um atributo derivado de outros atributos da entidade, um atributo que não faz sentido persistir, ou qualquer outra situação em que precisamos excluir o atributo do ciclo de vida de persistência da JPA.

Ela também pode ser aplicada a um método, como foi feito no exemplo do **isNew()** citado acima para indicar que ele não deve ser considerado durante o processo de persistência da JPA.

1.2. Classes de Entidades

Agora que já foram apresentadas as informações necessárias sobre a classe **AbstractPersistable,** será demonstrado como mapear duas entidades, as quais são base de exemplos para os repositórios que serão abordados durante este curso.

A primeira classe de entidade - **Listagem 3.1** - é a Contato, a qual possui o mapeamento para uma tabela no banco de dados chamada **Contatos.**

LISTAGEM 3.1: CLASSE DE ENTIDADE CONTATO

```
package com.curso.entity;

import java.util.Date;
import jakarta.persistence.*;
import org.springframework.data.jpa.domain.AbstractPersistable;

@Entity
@Table(name = "CONTATOS")
public class Contato extends AbstractPersistable<Long> {

   @Column(name = "nome", length = 64, nullable = false)
   private String nome;

   @Column(name = "idade")
   private Integer idade;

   @Column(name = "data_cadastro")
   @Temporal(TemporalType.DATE)
   private Date dtCadastro;

   @OneToOne(cascade = CascadeType.ALL)
   @JoinColumn(name = "endereco_id", nullable = false)
   private Endereco endereco;

   @Override
   public void setId(Long id) {
      super.setId(id);
   }

// demais métodos setters e getters foram omitidos nesta listagem.

}
```

Antes de analisar o código da classe Contato, é bom ressaltar que todas as anotações referentes ao mapeamento são do pacote **jakarta.persistence,** referente a especificação JPA. Porém caso você esteja usando a versão da JDK 11 o pacote será o **javax.persistence.** Mapear do pacote errado pode resultar em erros no projeto.

Observe também que a entidade herda as características de **AbstractPersistable.** Por este motivo, não é necessário ter na entidade um atributo **id,** nem os métodos **toString() , equals(), hashCode()** e **getId().**

Já o método **setId()** foi sobrescrito, o que é opcional, por ele ser originalmente, na superclasse, um método protegido. O que torna seu acesso restrito apenas à classe filha.

Outro ponto importante é informar, por meio de um *Generics,* na declaração de **AbstractPersistable,** qual o tipo de dado que representa o **id** da entidade. Neste caso, foi definido como um **Long**.

Como dito anteriormente, se você tem um conhecimento, mesmo que básico, sobre JPA ou Hibernate, vai entender perfeitamente este mapeamento. Entretanto, vou fazer uma rápida análise sobre o que há nesta classe:

- **@Entity** - esta anotação marca a classe como uma entidade gerenciada pelo framework ORM;
- **@Table** - tem algumas finalidades como: configurar o nome da tabela que a classe está mapeando *e* a adição de índices e *constraints;*
- **@Column** - sua função é indicar qual coluna na tabela está sendo mapeada. Caso o nome da coluna na tabela seja idêntico ao do atributo, a anotação não é necessária. Porém, é possível também adicionar informações complementares nesta anotação como: definir o tamanho da coluna, estabelecer se aceita valores nulos, entre outras opções;
- **@Temporal** - é uma anotação complementar para definir que um atributo é do tipo temporal (data e ou hora). Na tabela, a coluna também deve ser do tipo temporal (Date; DateTime; Timestamp);
- **@OneToOne** - esta anotação tem como finalidade mapear o relacionamento 1 - 1 entre entidades;
- **@JoinColumn** - indica qual o nome da chave estrangeira no relacionamento 1 - 1.

Se Contato tem um relacionamento 1 - 1**,** chegou o momento de conhecer o outro lado deste relacionamento, que é a entidade **Endereco**.

LISTAGEM 3.2: CLASSE DE ENTIDADE ENDERECO

```
package com.curso.entity;
import jakarta.persistence.*;
import org.springframework.data.jpa.domain.AbstractPersistable;

@Entity
@Table(name = "ENDERECOS")
public class Endereco extends AbstractPersistable<Long> {

   public enum TipoEndereco{
      RESIDENCIAL,  COMERCIAL
   }

   @Column(name = "tipo", length = 32, nullable = false)
   @Enumerated(EnumType.STRING)
   private TipoEndereco tipoEndereco;

   @Column(name = "logradouro", length = 64, nullable = false)
   private String logradouro;

   @Column(name = "cidade", length = 64, nullable = false)
   private String cidade;

   @Column(name = "estado", length = 2, nullable = false)
   private String estado;

   @Override
   public void setId(Long id) {
      super.setId(id);
   }

// demais métodos setters e getters foram omitidos nesta listagem.

}
```

Na classe Endereco, apresentada na **Listagem3.2,** há uma nova anotação em relação às abordadas na **Listagem3.1.** Sendo assim, vamos aprender um pouco mais sobre ela.

- **@Enumerated** - responsável pelo mapeamento de um atributo do tipo enum. Como parâmetro, esta anotação recebe o tipo de dado referente à coluna na tabela.

Agora já temos as duas classes de entidades devidamente mapeadas. Elas serão utilizadas como base para o estudo das operações de CRUD nos repositórios do Spring Data JPA.

1.3. Classes de Entidades - Sem AbstractPersistable

Uma outra forma de criar a classe entidade é sem utilizar o **AbstractPersistable.**

Segue abaixo a implementação ajustada de forma que o Id foi implementado na classe e não usamos o AbstractPersistable nesse exemplo.

LISTAGEM 3.3: CLASSE DE ENTIDADE CONTATO

```
package com.curso.entity;

import java.util.Date;
import jakarta.persistence.*;

@Entity
@Table(name = "CONTATOS")
public class Contato {

 @Id
 @GeneratedValue(strategy = GenerationType.IDENTITY)
 @Column(name = "cod_contato", nullable = false)
 private Long id;

 @Column(name = "nome", length = 64, nullable = false)
 private String nome;

 @Column(name = "idade")
 private Integer idade;

 @Column(name = "data_cadastro")
 @Temporal(TemporalType.DATE)
 private Date dtCadastro;

 @OneToOne(cascade = CascadeType.ALL)
 @JoinColumn(name = "cod_endereco", nullable = false)
 private Endereco endereco;

// demais métodos setters e getters foram omitidos nesta listagem.

}
```

LISTAGEM 3.4: CLASSE DE ENTIDADE ENDERECO

```
package com.curso.entity;
import jakarta.persistence.*;

@Entity
@Table(name = "ENDERECOS")
public class Endereco {

   public enum TipoEndereco{
      RESIDENCIAL,  COMERCIAL
   }

   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   @Column(name = "cod_endereco", nullable = false)
   private Long id;

   @Column(name = "tipo", length = 32, nullable = false)
   @Enumerated(EnumType.STRING)
   private TipoEndereco tipoEndereco;

   @Column(name = "logradouro", length = 64, nullable = false)
   private String logradouro;

   @Column(name = "cidade", length = 64, nullable = false)
   private String cidade;

   @Column(name = "estado", length = 2, nullable = false)
   private String estado;

// demais métodos setters e getters foram omitidos nesta listagem.

}
```

1.4. Sobre a anotação @GeneratedValue

A anotação **@GeneratedValue** é usada para especificar a estratégia de geração de valor para uma *chave primária* em uma entidade.

Em uma classe de entidade, geralmente é necessário definir **um atributo para representar a chave primária**, que é um identificador único para cada instância da entidade no banco de dados. A anotação **@GeneratedValue** é usada para indicar **como o valor da chave primária será gerado durante a persistência**.

ESTRATÉGIAS PARA GERAÇÃO DE VALORES

Existem diferentes estratégias disponíveis para a geração de valores de uma chave primária, e a anotação @GeneratedValue permite especificar qual estratégia será utilizada. Além disso, essa anotação é frequentemente usada **em conjunto com a anotação @Id**, como foi demonstrado na **LISTAGEM 3.3:** CLASSE DE ENTIDADE CONTATO, que indica que o atributo que recebe essa anotação é a chave primária da classe entidade.

Existem quatro estratégias de geração de valor disponíveis:

1. GenerationType.AUTO: Deixa a escolha da estratégia para o provedor JPA. O provedor JPA escolherá a estratégia apropriada com base no banco de dados que vamos conectar e na configuração realizada lá.

2. GenerationType.IDENTITY: Indica que o valor da chave primária será gerado pelo banco de dados usando recursos de **autoincremento**. Essa estratégia é adequada para bancos de dados que tenham suas tabelas criadas usando essa definição.

```
package com.curso.entity;
import jakarta.persistence.*;

@Entity
@Table(name = "ENDERECOS")
public class Endereco {

   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   @Column(name = "cod_endereco", nullable = false)
   private Long id;
   // outros atributos e método

}
```

No exemplo acima, a anotação **@GeneratedValue(strategy = GenerationType.IDENTITY)** é usada em conjunto com a anotação **@Id** para indicar que o valor da chave primária id será gerado pelo banco de dados usando a estratégia de **autoincremento**.

Essa anotação fornece uma maneira de configurar a geração de valores de chave primária no JPA, permitindo que você escolha a estratégia adequada com base nas características do banco de dados que está sendo utilizado.

3. GenerationType.SEQUENCE: Indica que o valor da chave primária será gerado por uma sequência definida no banco de dados. Essa estratégia é adequada para bancos de dados que suportam sequências, como Oracle e PostgreSQL. É importante ressaltar que no **MariaDB**, versão 10.3 e posterior, há suporte para o uso de sequências através da funcionalidade **SEQUENCE**. No entanto, é importante observar que, por padrão, o MariaDB não possui uma implementação nativa de sequências.

No **MariaDB**, a **opção mais comum para a geração de valores de chave primária é utilizar a estratégia de autoincremento**, por meio da propriedade **AUTO_INCREMENT**. Essa propriedade permite que a coluna da chave primária seja incrementada automaticamente pelo banco de dados ao inserir novos registros.

CREATE TABLE ENDERECOS (

cod_endereco INTEGER AUTO_INCREMENT PRIMARY KEY,

tipo VARCHAR(32) NOT NULL,

logradouro VARCHAR(64) NOT NULL,

cidade VARCHAR(64) NOT NULL,

estado CHAR(2) NOT NULL

);

No exemplo acima, a coluna **cod_endereco** é definida com a propriedade **AUTO_INCREMENT**, indicando que o MariaDB irá gerar automaticamente valores sequenciais para essa coluna. A nossa implementação da classe entidade deve contemplar as regras de nossa tabela "ENDERECOS" criada no banco de dados, e usamos para isso o recurso das anotações que estão sendo apresentadas durante os estudos da

disciplina. Nesse exemplo teríamos de usar o **GenerationType.IDENTITY** para contemplar a solução.

Portanto, **se você está utilizando o MariaDB** e deseja gerar valores de chave primária usando sequências, é necessário criar uma sequência personalizada e utilizar recursos adicionais, como gatilhos (triggers) lá no banco de dados, para implementar a funcionalidade de sequência.

Mas e se o projeto que vamos trabalhar define o uso da estratégia de sequências para gerar a chave primária e não queremos ter todo esse trabalho de usar recursos adicionais no banco de dados?

Nesse caso nós poderíamos optar pelo uso do banco de dados PostgreSql por exemplo, que já possui esse recurso de forma nativa.

É muito importante observar que a configuração da sequência pode variar de acordo com o banco de dados que você está utilizando. Consulte a documentação do banco de dados específico que será utilizado em seu futuro projeto para obter detalhes sobre como criar e configurar sequências para a geração de valores de chave primária.

Para fins de aprendizado, seguiremos exemplificando o MariaDB para abordar tudo que foi planejado para o conteúdo da disciplina. Nesse sentido vamos usar uma abordagem mais simples, que é a estratégia de **autoincremento** com a propriedade **AUTO_INCREMENT** para a geração de valores de chave primária no **MariaDB, fazendo de acordo com o exemplo mostrado logo acima.**

4. GenerationType.TABLE: Indica que o valor da chave primária será gerado usando uma tabela especial no banco de dados para controlar a geração de valores. Essa estratégia é menos comum e é usada principalmente em casos onde as estratégias de autoincremento ou sequência não estão disponíveis.

Mas afinal, quem é o responsável por gerar o valor da chave primária, independente do tipo de estratégia que informamos?

As estratégias de geração do valor da chave primária informadas no **@GeneratedValue**, informam ao provedor JPA como tratar a geração do valor da chave primária durante as operações de persistência. O provedor JPA, por sua vez, **delega** a responsabilidade real de criar a nova chave primária ao banco de dados.

As anotações, como **@GeneratedValue(strategy = GenerationType.IDENTITY)** ou **@GeneratedValue(strategy = GenerationType.SEQUENCE)**, são utilizadas para indicar ao provedor JPA a estratégia desejada para a geração do valor da chave primária. Essas estratégias incluem o uso de recursos fornecidos pelo banco de dados, como colunas autoincrementáveis ou sequências.

No caso da estratégia **GenerationType.IDENTITY**, conforme foi demonstrado acima, o provedor JPA solicita ao banco de dados que utilize sua funcionalidade de **autoincremento** para gerar valores sequenciais para a chave primária durante a inserção de registros.

Já no caso da estratégia **GenerationType.SEQUENCE**, o provedor JPA solicita ao banco de dados que utilize **uma sequência definida** para gerar valores únicos para a chave primária durante a inserção de registros.

Dessa forma, o banco de dados é responsável por gerar os valores da chave primária de acordo com a estratégia configurada. **O JPA então obtém esses valores do banco de dados** e os atribui aos atributos da chave primária nas entidades correspondentes.

Sendo assim, **as anotações** referentes a estratégia para geração do valor da chave primária no JPA **informam ao provedor JPA como lidar com a geração do valor da chave primária**, mas é o banco de dados que realiza a geração real desses valores com base na estratégia definida.

E o que é o PROVEDOR JPA?

O provedor JPA é uma implementação da especificação **Java Persistence API (JPA)**. Conforme já conversamos na disciplina a **JPA é uma API** padrão do Java que define uma interface comum **para mapeamento objeto-relacional** e operações de persistência em bancos de dados relacionais.

A especificação JPA descreve a funcionalidade que um provedor JPA deve implementar para permitir a integração entre aplicativos Java e bancos de dados relacionais de maneira padronizada. **O provedor JPA é responsável por fornecer a implementação concreta das classes e métodos definidos pela especificação**.

Conforme citado anteriormente em nossa disciplina, **existem vários provedores JPA disponíveis, como Hibernate, EclipseLink, OpenJPA, entre outros.** Cada provedor JPA possui sua própria implementação, recursos extras e peculiaridades, **mas todos eles devem aderir às interfaces e especificações definidas pela JPA**.

Ao desenvolver um aplicativo que utiliza JPA, você precisa configurar o provedor JPA escolhido no seu projeto. Isso geralmente envolve a inclusão da biblioteca do provedor JPA no classpath do projeto e a configuração das propriedades específicas do provedor, como informações de conexão com o banco de dados e outras configurações de mapeamento.

Uma vez configurado o provedor JPA, você pode usar as anotações e recursos fornecidos pela JPA para mapear suas classes de entidade, executar operações de persistência (inserir, atualizar, excluir) e realizar consultas em bancos de dados relacionais usando uma abordagem orientada a objetos.

Para concluir, entendemos que o provedor JPA é a implementação concreta que permite a interação entre seu aplicativo Java e o banco de dados relacional, seguindo os padrões e especificações definidos pela Java Persistence API.